CS178 25W
Prof. Berg

03/15/25
Julian Zhang

Final Project: Toxic Comments Challenge

In order to demonstrate advanced understanding of Machine Learning and Data Mining techniques, I have selected the Toxic Comments Dataset from Kaggle to train a model to classify similar data. The dataset is a collection of comments on online forums made available by the Wikimedia Foundation, some of which contain toxic, obscene, and/or hateful language.

1. Exploring the dataset

The dataset is divided into files ***train.csv***, ***test.csv***, ***test_labels.csv***. To train the model, we will be using data from ***train.csv***, which consists of **159,571** data points of string text of length < 5000 characters (**X_tr**) and their binary label vectors of length **6** (**y_tr**). **63,978** data points from ***test.csv*** (**X_te**) are then fed into the model to predict output labels, which will be graded against the correct labels provided in ***test_labels.csv*** (**y_te**).

```
print(X_tr.shape)
print(y_tr.shape)
print(X_te.shape)
print(y_te.shape)
print(X_tr.dtype)
```

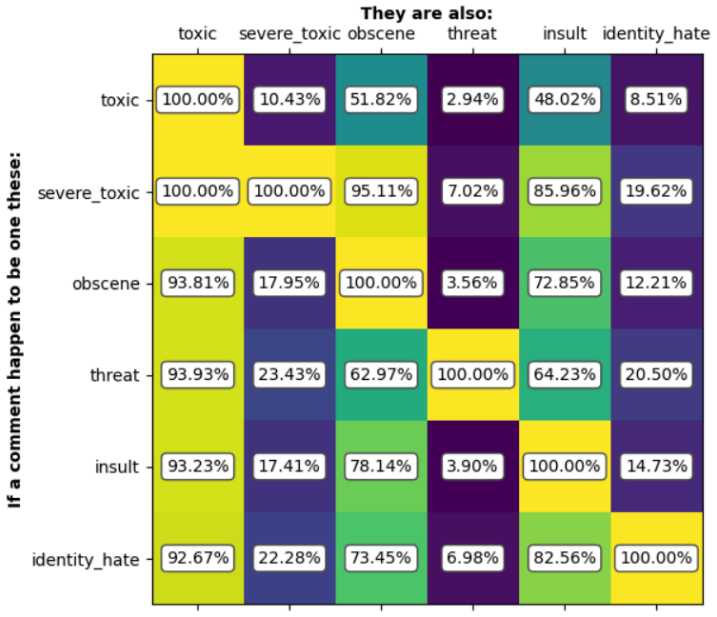
✓ 0.8s

```
(159571,)
(159571, 6)
(63978,)
(63978, 6)
<U5000
```

A quick proportion statistic calculation shows that a small percentage of comments are classified under any of the toxic labels:

```
label names: ['toxic' 'severe_toxic' 'obscene' 'threat' 'insult' 'identity_hate']
9.584% comments in the dataset are toxic
0.9996% comments in the dataset are severe_toxic
5.295% comments in the dataset are obscene
0.2996% comments in the dataset are threat
4.936% comments in the dataset are insult
0.8805% comments in the dataset are identity_hate
```

However, once a comment is classified under any of the toxic labels, they become very likely to be classified under many other toxic labels as well.
<u>Example for reading this matrix:</u>
*If a comment is **severe_toxic**, it is also **obscene** in 95.11% of the cases and **insult** in 85.96% of the cases.*



This behavior of the data exhibits an information gain in the form of decreasing Shannon entropy after being split by features.

As an additional point of complexity, the input of this dataset is not a numeric feature vector but a text string of differing length. Therefore a **tokenizer** is required to transform the strings into a numeric feature array. I am using sklearn's **TfidfVectorizer** for this purpose. It becomes evident however, that strings that contain different words happen to have different sizes in their transformed vector. This is due to the fact that the vectorizers count occurrences of words. The vectorizer is therefore fitted on the training data to produce identical dimensions on testing data.

2. Model/Algorithm Selection

In the previous section, we have observed information gain from feature splitting, this makes the **Decision Tree** model suitable for this dataset. A quick evaluation on the first 10,000 data points achieved an impressive 4.4% error rate, which is below the common acceptable rate for text classification (5%). Upgrading to a **RandomForestClassifier** reduced the error rate further down to 3.7%.

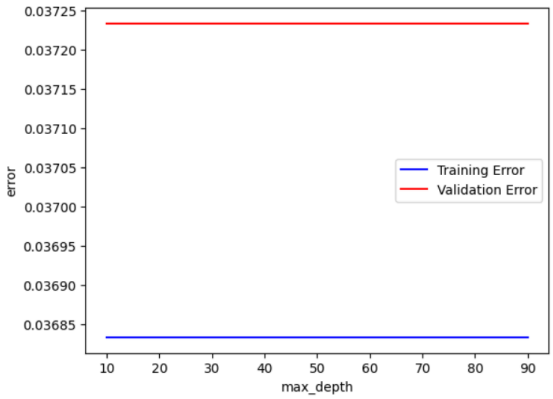
```
from sklearn.tree import DecisionTreeClassifier

model = DecisionTreeClassifier().fit(X_tr_vec, y_tr[:LIMIT])
print("Error:", (model.predict(X_te_vec) != y_te[:LIMIT]).mean())
```

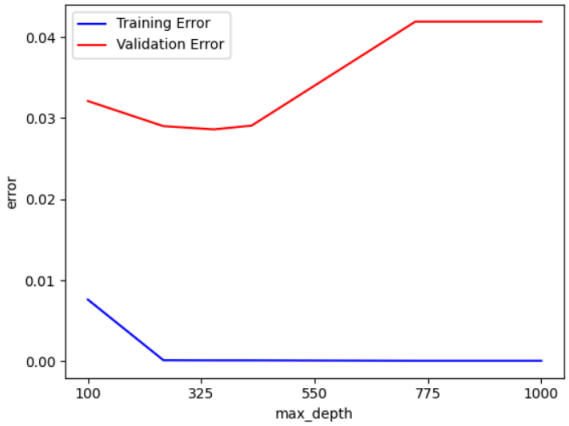
```
Error: 0.04393333333333333
```

The next logical step was to vary the hyperparameters in order to find the optimal degree of fitting. However, this was when problems were encountered:

A multitude of hyperparameters (max_depth, n_estimators, min_samples_leaf, criterion) were adjusted in order to find fluctuations in error rate, but none were observed.
Upon closer inspection, the input feature vectors are very long and complex: Each of the 10,000 data points used to evaluate preliminary model performances are transformed by the vectorizer into a feature vector with length 36,382. There are simply too many features for a small decision tree to split and classify. Additionally, the text-extracted feature vectors are very sparse, which causes inefficiency with space.



Operating on that information, the max_depth parameter was greatly increased in orders of magnitude. We can finally observe the familiar error curve, which corresponds to an optimal value of max_depth = 250, any higher will cause overfitting.



The next strategy was to use the brute-force power of Perceptrons in **MLPClassifier** to overcome the complexity of the inputs. This amounted to little improvement however, as it took 30 minutes to complete 500 epochs of gradient

descent, and the final error was no better than that of the Random Forest, having failed to converge by the 400th epoch:

```
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier
scaler = StandardScaler().fit(X_tr_vec)
X_tr_trans = scaler.transform(X_tr_vec)
X_te_trans = scaler.transform(X_te_vec)

model = MultiOutputClassifier(MLPClassifier(hidden_layer_sizes=(100,), solver="sgd", learning_rate_init=0.001, max_iter=500))
model.fit(X_tr_vec, y_tr[:LIMIT])
print("Error:", (model.predict(X_te_vec) != y_te[:LIMIT]).mean())
✓ 30m 8.1s
c:\Users\julia\Documents\Workspace\CS178\.venv\Lib\site-packages\sklearn\neural_network\_multilayer_perceptron.py:691: Convergence
  warnings.warn(
Error: 0.037233333333333334
```

The performance of some classic classifier models are then evaluated, here is a table of all the models that have been attempted so far:

Model	Validation Error	Time	Parameters altered
DecisionTreeClassifier	4.39%	51.2s	criterion, max_depth, min_samples_leaf
RandomForestClassifier	2.9%	23.4s	criterion, max_depth, min_samples_leaf, n_estimators
ComplementNB	4.21%	43.9s	
MLPClassifier	3.72%	31m	LR, max_iter, hidden_layer_sizes
SVM	Abandoned	Too long	
KNeighborsClassifier	3.61%	5.5m	k
GradientBoostingClassif	Abandoned	Too long	

NOTE: Some of these classifiers are used in conjunction with **MultiOutputClassifier**, as they are not suitable for multi-target training by themselves.

So far, the best performing models are the RandomForestClassifier and KNeighborsClassifier. They will be evaluated in section 4 using the entire training set and via another method, discussed below:

3. Additional Depth

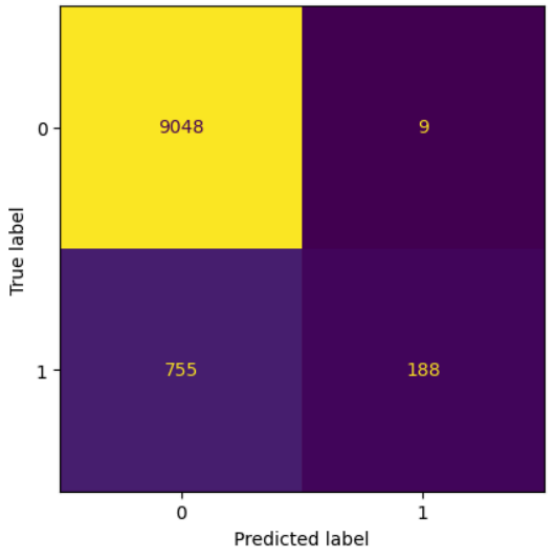
In researching NLP and text classification, I have discovered that error rates are not the best metric to use for toxic comment classification. Chasing a low error rate on training and validation can actually cause a higher rate of false positives when the model is deployed for use. Instead, we could use Precision metrics, which penalizes false positives greatly, or Macro-averaged AUC (Area Under Curve), which is suitable for imbalanced classes like in this

dataset (0.88% identity_hate vs 9.58% toxic). Applying sklearn's **roc_auc** function on the predicted probabilities of the model yields a mediocre score of 0.56 averaged across all features, which is just slightly better than pure randomness.
Sklearn's built in classification_report enables a more detailed look on the model accuracy:

	precision	recall	f1-score	support
toxic	0.95	0.20	0.33	943
severe_toxic	0.00	0.00	0.00	56
obscene	0.99	0.16	0.27	565
threat	0.00	0.00	0.00	37
insult	0.94	0.06	0.12	511
identity_hate	0.00	0.00	0.00	122
micro avg	0.96	0.14	0.24	2234
macro avg	0.48	0.07	0.12	2234
weighted avg	0.87	0.14	0.24	2234
samples avg	0.02	0.01	0.01	2234

We can see that the **precision** rate is quite high, meaning that the model is well suited against false positives. However there is a high amount of false negatives (**recall**), which lowers the overall **f1-score**. We can also see that some features (e.g. severe_toxic) have missing metrics: This is because they have such a low rate of occurrence in the limited training dataset (**support**), that the model failed to learn those features entirely, and therefore would only predict zeros.

A **confusion matrix** is also helpful to visualize the classification:
We can see that the model makes a very small amount of false positives, but the large amount of false negatives dwarfs even the true negatives.
One possible solution to this issue is to use a skewed dataset with more data points that are labeled (with '1's) than otherwise.
The **imblearn** library provides a similar solution by implementing over-sampling and under-sampling techniques, which makes the model more favorable to minority labels.
Unfortunately, imblearn does not support multi-output targets, therefore I will attempt to manually recreate under-sampling:



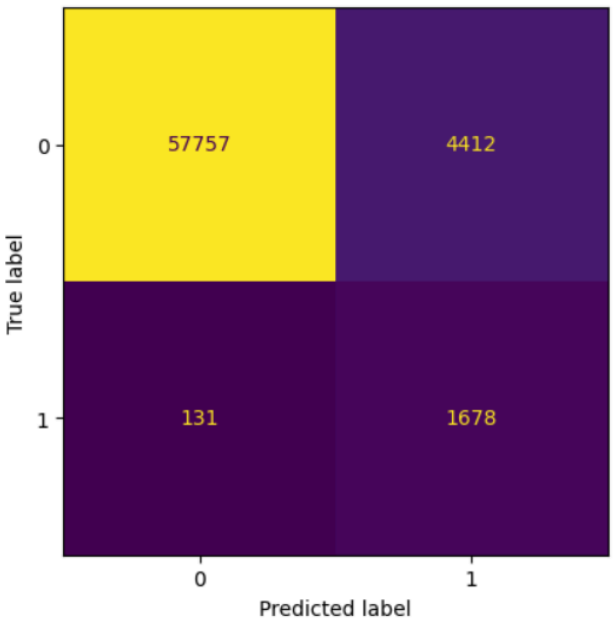
```
(y_tr[:LIMIT] == np.array([0]*6)).mean()
✓ 0.0s
np.float64(0.9633412921729722)
```

Over 96% of the train data have all labels set to 0. We could simply remove a portion of them to over-represent the minority data that has any label set to 1.

```
zero_idx = np.where(np.all(y_tr == np.array([0]*6), axis=1))[0]
remove_idx = np.random.choice(zero_idx, size=50000, replace=False)
y_tr_trim = np.delete(y_tr, remove_idx, axis=0)

mask = np.ones(X_tr_vec.shape[0], dtype=bool)
mask[remove_idx] = False
X_tr_trim = X_tr_vec[mask]
print(X_tr_trim.shape)
print(y_tr_trim.shape)
✓ 0.0s
(109571, 189775)
(109571, 6)
```

Here we have randomly removed 50,000 entries from the train data that have label vectors of 0. Training with this trimmed dataset yielded a slightly better error at 3.12%.



The confusion matrix shows a marked improvement in false negatives. However we can observe that trimming the train data greatly increases the number of false positives, which could have negative repercussions, e.g. forum users being unjustly banned; this is a tradeoff that I believe shouldn't be made.

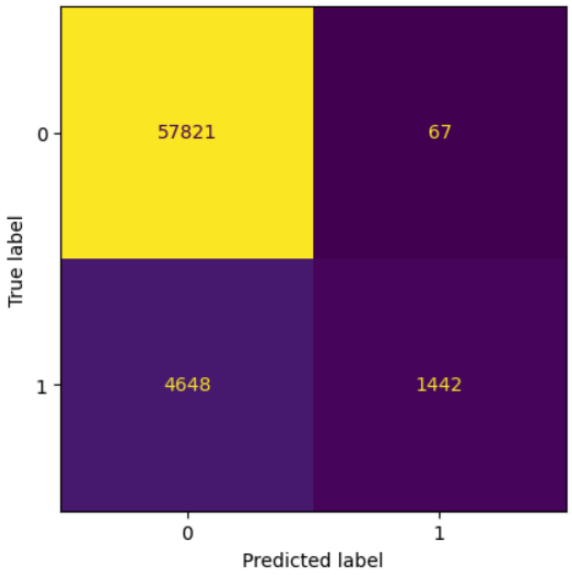
The AUC shows a slight improvement at an average 0.61 across all features.

4. Performance validation

Training the RandomForestClassifier with all 159,571 of the data points took 4 minutes, and achieved an error rate of 3.18%, which is an optimal value from fine-tuning the parameters in section 2.

	precision
toxic	0.96
severe_toxic	0.97
obscene	0.98
threat	0.97
insult	0.98
identity_hate	0.97

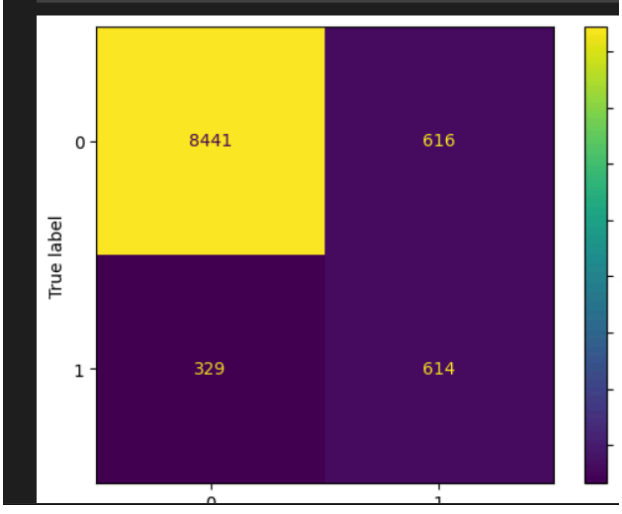
The minority labels finally have enough samples to be predicted, they show a similarly high amount of precision.



The confusion matrix displays the expected arrangement of predictions, with most correct predictions being true negatives, and most incorrect predictions being false negatives. The model's most prominent feature is having a very low amount of false positives.

Training the KNeighborsClassifier in conjunction with MultiOutputClassifier took 4 and a half minutes, and achieved an error rate of 3.61%, which is the optimal value found at K=4. The classification report and confusion matrix display an average performance across the map, with approximately the same amount of true positives and false positives. The model's most prominent feature is having a low amount of false negatives.

toxic	0.50	0.65	0.57	943
severe_toxic	0.10	0.27	0.15	56
obscene	0.69	0.42	0.52	565
threat	0.27	0.08	0.12	37
insult	0.60	0.34	0.43	511
identity_hate	0.52	0.26	0.35	122
micro avg	0.52	0.48	0.50	2234
macro avg	0.45	0.34	0.36	2234
weighted avg	0.56	0.48	0.50	2234
samples avg	0.05	0.05	0.05	2234



From the performance analysis of these two models, I conclude that they can be used with discretion in different applications: The Random Forest could be applied to anti-toxicity enforcement, where the low chance of false positives can guarantee a robust amount of fairness. Whereas the Neighbors classifier could be used in general purpose censorship, where it's important to prevent toxic content from slipping through the cracks.